Dynamic Planar Point Location in External Memory

J. Ian Munro
Cheriton School of Computer Science, University of Waterloo, Canada
imunro@uwaterloo.ca

Yakov Nekrich
Cheriton School of Computer Science, University of Waterloo, Canada
yakov.nekrich@googlemail.com

Abstract

In this paper we describe a fully-dynamic data structure for the planar point location problem in the external memory model. Our data structure supports queries in $O(\log_B n(\log\log_B n)^3))$ I/Os and updates in $O(\log_B n(\log\log_B n)^2))$ amortized I/Os, where n is the number of segments in the subdivision and B is the block size. This is the first dynamic data structure with almost-optimal query cost. For comparison all previously known results for this problem require $O(\log_B^2 n)$ I/Os to answer queries. Our result almost matches the best known upper bound in the internal-memory model.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Theory of computation

Keywords and phrases Data Structures, Dynamic Data Structures, Planar Point Location, External Memory

Digital Object Identifier 10.4230/LIPIcs.SoCG.2019.52

Related Version A full version of the paper is available at `http://arxiv.org/abs/1903.06601`.

1 Introduction

Planar point location is a classical computational geometry problem with a number of important applications. In this problem we keep a polygonal subdivision Π of the two-dimensional plane in a data structure; for an arbitrary query point q, we must be able to find the face of Π that contains q. In this paper we study the dynamic version of this problem in the external memory model. We show that a planar subdivision can be maintained under insertions and deletions of edges, so that the cost of queries and updates is close to $O(\log_B n)$, where n is the number of segments in the subdivision and B is the block size.

Planar point location problem was studied extensively in different computational models. Dynamic internal-memory data structures for general subdivisions were described by Bentley [11], Cheng and Janardan [16], Baumgarten et al. [9], Arge et al. [3], and Chan and Nekrich [13]. Table 1 lists previous results. We did not include in this table many other results for special cases of the point location problem, such as the data structures for monotone, convex, and orthogonal subdivisions, e.g., [25, 26, 18, 17, 21, 20, 14]. The currently best data structure [13] achieves[1] $O(\log n)$ query time and $O(\log^{1+\varepsilon} n)$ update time or $O(\log^{1+\varepsilon} n)$ query time and $O(\log n)$ update time; the best query-update trade-off described in [13] is $O(\log n \log\log n)$ randomized query time and $O(\log n \log\log n)$ update time. See Table 1.

[1] In this paper $\log n$ denotes the binary logarithm of n when the logarithm base is not specified.

35th International Symposium on Computational Geometry (SoCG 2019).
Editors: Gill Barequet and Yusu Wang; Article No. 52; pp. 52:1–52:15

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

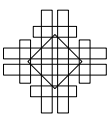

Table 1 Previous results on dynamic planar point location in internal memory. Entries marked † and ‡ require amortization and (Las Vegas) randomization respectively, $\varepsilon > 0$ is an arbitrarily small constant. Results marked * are in the RAM model, all other results are in the pointer machine model. Space usage is measured in words.

Reference	Space	Query Time	Insertion Time	Deletion Time	
Bentley [11]	$n \log n$	$\log^2 n$	$\log^2 n$	$\log^2 n$	
Cheng–Janardan [16]	n	$\log^2 n$	$\log n$	$\log n$	
Baumgarten et al. [9]	n	$\log n \log\log n$	$\log n \log\log n$	$\log^2 n$	†
Arge et al. [3]	n	$\log n$	$\log^{1+\varepsilon} n$	$\log^{2+\varepsilon} n$	†
Arge et al. [3]	n	$\log n$	$\log n(\log\log n)^{1+\varepsilon}$	$\log^2 n/\log\log n$	†‡*
Chan and Nekrich [13]	n	$\log n(\log\log n)^2$	$\log n \log\log n$	$\log n \log\log n$	
Chan and Nekrich [13]	n	$\log n$	$\log^{1+\varepsilon} n$	$\log^{1+\varepsilon} n$	
Chan and Nekrich [13]	n	$\log n$	$\log^{1+\varepsilon} n$	$\log n(\log\log n)^{1+\varepsilon}$	*
Chan and Nekrich [13]	n	$\log^{1+\varepsilon} n$	$\log n$	$\log n$	
Chan and Nekrich [13]	n	$\log n \log\log n$	$\log n \log\log n$	$\log n \log\log n$	‡*

In the external memory model [2] the data can be stored in the internal memory of size M or on the external disk. Arithmetic operations can be performed only on data in the internal memory. Every input/output operation (I/O) either reads a block of B contiguous words from the disk into the internal memory or writes B words from the internal memory into disk. Measures of efficiency in this model are the number of I/Os needed to solve a problem and the amount of used disk space.

Goodrich et al. [22] presented a linear-space static external data structure for point location in a monotone subdivision with $O(\log_B n)$ query cost. Arge et al. [5] designed a data structure for a general subdivison with the same query cost. Data structures for answering a batch of point location queries were considered in [22] and [8]. Only three external-memory results are known for the dynamic case. The data structure of Agarwal, Arge, Brodal, and Vitter [1] supports queries on monotone subdivisions in $O(\log_B^2 n)$ I/Os and updates in $O(\log_B^2 n)$ I/Os amortized. Arge and Vahrenhold [7] considered the case of general subdivisons; they retain the same cost for queries and insertions as [1] and reduce the deletion cost to $O(\log_B n)$. Arge, Brodal, and Rao [4] reduced the insertion cost to $O(\log_B n)$. Thus all previous dynamic data structures did not break $O(\log_B^2 n)$ query cost barrier. For comparison the first internal-memory data structure with query time close to logarithmic was presented by Baumgarten et al [9] in 1994. See Table 2. All previous data structures use $O(n)$ words of space (or $O(n/B)$ blocks of B words[2]).

In this paper we show that it is possible to break the $O(\log_B^2 n)$ barrier for the dynamic point location problem. Our data structure answers queries in $O(\log_B n(\log\log_B n)^3)$ I/Os, supports updates in $O(\log_B n(\log\log_B n)^2)$ I/Os amortized, and uses linear space. Thus we achieve close to logarithmic query cost and a query-update trade-off almost matching the state-of-the-art upper bounds in the internal memory model. Our result is within double-logarithmic factors from optimal. Additionally we describe a data structure that supports point location queries in an orthogonal subdivision with $O(\log_B n \log\log_B n)$ query cost and $O(\log_B n \log\log_B n)$ amortized update cost. The computational model used in this paper is the standard external memory model [2].

[2] Space usage of external-memory data structures is frequently measured in disk blocks of B words. In this paper we measure the space usage in words. But the space usage of $O(n)$ words is equivalent to $O(n/B)$ blocks of space.

Table 2 Previous and new results on dynamic planar point location in external memory. G denotes most general subdivisions, M denotes monotone subdivision, and O denotes orthogonal subdivision. Space usage is measured in words and update cost is amortized.

Reference	Space	Query Cost	Insertion Cost	Deletion Cost	
Agarwal et al [1]	n	$\log_B^2 n$	$\log_B^2 n$	$\log_B^2 n$	M
Arge and Vahrenhold [7]	n	$\log_B^2 n$	$\log_B^2 n$	$\log_B n$	G
Arge et al [4]	n	$\log_B^2 n$	$\log_B n$	$\log_B n$	G
This paper	n	$\log_B n(\log\log_B n)^3$	$\log_B n(\log\log_B n)^2$	$\log_B n(\log\log_B n)^2$	G
This paper	n	$\log_B n \log\log_B n$	$\log_B n \log\log_B n$	$\log_B n \log\log_B n$	O

2 Overview

2.1 Overall Structure

As in the previous works, we concentrate on answering *vertical ray shooting* queries. The successor segment of a point q in a set S of non-intersecting segments is the first segment that is hit by a ray emanating from q in the $+y$-direction. Symmetrically, the predecessor segment of q in S is the first segment hit by a ray emanating from q in the $-y$ direction. A vertical ray shooting query for a point q on a set of segments S asks for the successor segment of q in S. If we know the successor segment or the predecessor segment of q among all segments of a subdivision Π, then we can answer a point location query on Π (i.e., identify the face of Π containing q) in $O(\log_B n)$ I/Os [7]. In the rest of this paper we will show how to answer vertical ray shooting queries on a dynamic set of non-intersecting segments.

Our base data structure is a variant of the segment tree. Let $\mathcal{S}$ be a set of segments. We store a tree $\mathcal{T}$ on x-coordinates of segment endpoints. Every leaf contains $\Theta(B)$ segment endpoints and every internal node has $r = \Theta(B^\delta)$ children for $\delta = 1/8$. Thus the height of $\mathcal{T}$ is $O(\log_B n)$. We associate a vertical slab with every node u of $\mathcal{T}$. The slab of the root node is $[x_{\min}, x_{\max}] \times \mathbb{R}$, where $x_{\min}$ and $x_{\max}$ denote the x-coordinates of the leftmost and the rightmost segment endpoints. The slab of an internal node u is divided into $\Theta(B^\delta)$ slabs that correspond to the children of u. A segment s *spans* the slab of a node u (or simply spans u) if it crosses its vertical boundaries.

A segment s is assigned to an internal node u, if s spans at least one child u_i of u but does not span u. We assign s to a leaf node ℓ if at least one endpoint of s is stored in ℓ. All segments assigned to a node u are trimmed to slab boundaries of children and stored in a *multi-slab* data structure $C(u)$: Suppose that a segment s is assigned to u and it spans the children $u_f, \ldots, u_l$ of u. Then we store the segment $s_u = [p_f, p_l]$ in $C(u)$, where p_f is the point where s intersect the left slab boundary of u_f and p_l is the point where s intersects the right boundary of u_l. See Fig. 1. Each segment is assigned to $O(\log_B n)$ nodes of $\mathcal{T}$.

In order to answer a vertical ray shooting query for a point q, we identify the leaf ℓ such that the slab of ℓ contains q. Then we visit all nodes u on the path π_ℓ from the root of $\mathcal{T}$ to ℓ and answer vertical ray shooting queries in multi-slab structures $C(u)$.

2.2 Our Approach

Thus our goal is to answer $O(\log_B n)$ ray shooting queries in multi-slab structures along a path in the segment tree $\mathcal{T}$ with as few I/Os as possible. Segments stored in a multi-slab are not comparable in the general case; see Fig. 2. It is possible to impose a total order $\prec$ on all segments in the following sense: let l be a vertical line that intersects segments s_1 and s_2; if the intersection of l with s_1 is above the intersection of l with s_2, then $s_2 \prec s_1$.

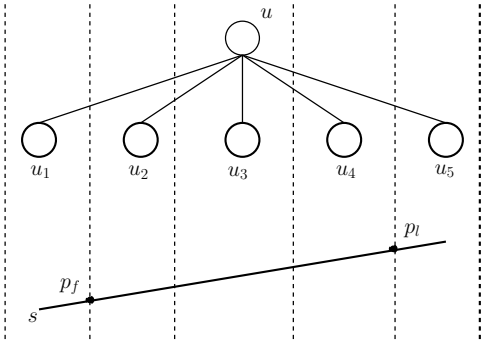

Figure 1 Segment s is assigned to node u. The trimmed segment $[p_f, p_l]$ is stored in $C(u)$.

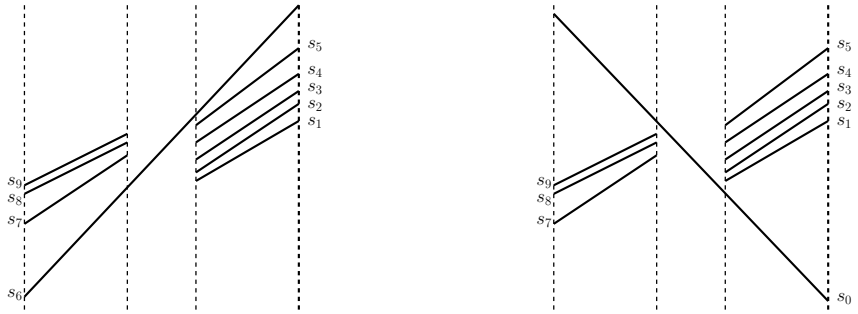

Figure 2 Left: example of segment order in a multi-slab; $s_1 \prec s_2 \prec s_3 \prec s_4 \prec s_5 \prec s_6 \prec s_7 \prec s_8 \prec s_9$. Right: a deletion and an insertion of one new segment in a multi-slab changes the order of segments to $s_7 \prec s_8 \prec s_9 \prec s_0 \prec s_1 \prec s_2 \prec s_3 \prec s_4 \prec s_5$.

We can find such a total order in $O((K/B)\log_{M/B} K)$ I/Os, where K is the number of segments [8, Lemma 3]. But this ordering is not stable under updates: even a single deletion and a single insertion can lead to significant changes in the order of segments. See Fig. 2. Therefore it is hard to apply standard techniques, such as fractional cascading [15, 23], in order to speed-up ray shooting queries. Previous external-memory solutions in [1, 4] essentially perform $O(\log_B n)$ independent searches in the nodes of a segment tree or an interval tree in order to answer a query. Each search takes $O(\log_B n)$ I/Os, hence the total query cost is $O(\log_B^2 n)$.

Internal memory data structures achieve $O(\log n)$ query cost using dynamic fractional cascading [15, 23]. Essentially the difference with external memory is as follows: since we aim for $O(\log_2 n)$ query cost in internal memory, we can afford to use base tree $\mathcal{T}$ with small node degree. In this special case the segments stored in sets $C(u)$, $u \in \mathcal{T}$, can be ordered resp. divided into a small number of ordered sets. When the order of segments in $C(u)$ is known, we can apply the fractional cascading technique [15, 23] to speed up queries. Unfortunately dynamic fractional cascading does not work in the case when the total order of segments in $C(u)$ is not known. Hence we cannot use previous internal memory solutions of the point location problem [16, 9, 3, 13] to decrease the query cost in external memory.

In this paper we propose a different approach. Searching in a multi-slab structure $C(u)$ is based on a weighted search among segments of $C(u)$. Weights of segments are chosen in such way that the total cost of searching in all multi-slab structures along a path π is logarithmic. We also use fractional cascading, but this technique plays an auxiliary role: we

apply fractional cascading to compute the weights of segments and to navigate between the tree nodes. Interestingly, fractional cascading is usually combined with the union-split-find data structure, which is not used in our construction.

This paper is structured as follows. In Section 3 we show how our new technique, that will be henceforth called weighted telescoping search, can be used to solve the static vertical ray shooting problem. Next we turn to the dynamic case. In our exposition we assume, for simplicity, that the set of segment x-coordinates is fixed, i.e., the tree $\mathcal{T}$ does not change. We also assume that the block size B is sufficiently large, $B > \log^8 n$. We show how our static data structure from Section 3 can be modified to support insertions in Section 4. To maintain the order of segments in a multi-slab under insertions we pursue the following strategy: when a new segment is inserted into the multi-slab structure $C(u)$, we split it into a number of *unit* segments, such that every unit segment spans exactly one child of u. Unit segments can be inserted into a multi-slab so that the order of other segments is not affected. The number of unit segments per inserted segment can be large; however we can use buffering to reduce the cost of updates.[3] We need to make some further changes in our data structure in order to support deletions; the fully-dynamic solution for large B is described in Section 5. The main result of Section 5, summed up in Lemma 2, is the data structure that answers queries in $O(\log_B n \log\log_B n)$ I/Os; insertions and deletions are supported in $O(\log_B^2 n)$ and $O(\log_B n)$ amortized I/Os respectively. We show how to reduce the cost of insertions in Section 6. We address some missing technical details and consider the case of small block size B in Section 7. In the full version of this paper [24] we will show how the space usage can be reduced to linear and address some issues related to updates of bridge segments. The special case of vertical ray shooting among horizontal segments is also considered in the full version.

3 Ray Shooting: Static Structure

In this section we show how the weighted telescoping search can be used to solve the static point location problem. Let $\mathcal{T}$ be the tree, defined in Section 2.1, with node degree $r = B^\delta$ for $\delta = 1/8$. Let $C(u)$ be the set of segments that span at least one child of u but do not span u.

Augmented Catalogs. We keep augmented catalogs $AC(u) \supset C(u)$ in every node u. Each $AC(u)$ is divided into subsets $AC_{ij}(u)$ for $1 \le i \le j \le r$; $AC_{ij}(u)$ contains segments that span children $u_i, \ldots, u_j$ of u and only those children. Augmented catalogs $AC(u)$ satisfy the following properties:

- **(i)** If a segment $s \in (AC(u) \setminus C(u))$, then $s \in C(v)$ for an ancestor v of u and s spans u.
- **(ii)** Let $E_i(u) = AC(u) \cap AC(u_i)$ for a child u_i of u. For any f and l, $f \le i \le l$, there are at most $d = O(r^4)$ elements of $AC_{fl}(u)$ between any two consecutive elements of $E_i(u)$.
- **(iii)** If $i \neq j$, then $E_i(u) \cap E_j(u) = \emptyset$.

Elements of $E_i(u)$ for some $1 \le i \le r$ will be called down-bridges; elements of the set $UP(u) = AC(u) \cap AC(par(u))$, where $par(u)$ denotes the parent node of u, are called up-bridges. We will say that a sub-list of a catalog $AC(u)$ bounded by two up-bridges is a *portion* of $AC(u)$. We refer to e.g., [3] or [13] for an explanation how we can construct

[3] As a side remark, this approach works with weighted telescoping search, but it would not work with the standard fractional cascading used in internal-memory solutions [16, 9, 3, 13]. The latter technique relies on a union-split-find data structure (USF) and it is not known how to combine buffering with USF.

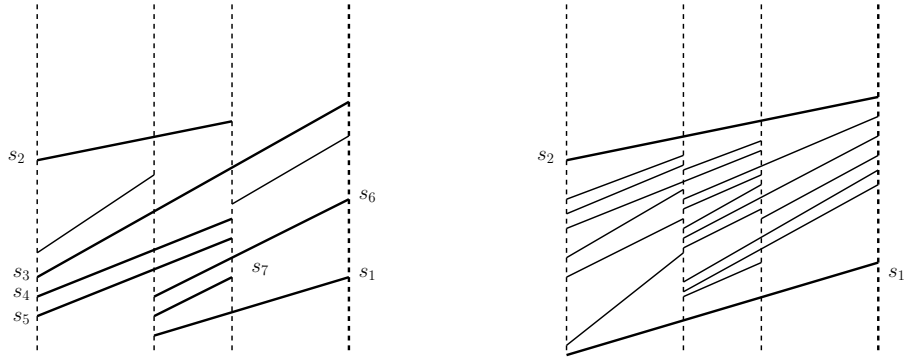

Figure 3 Computation of segment weights. Left: segments s_1 and s_2 are down-bridges from $E_2(u)$. Segments s_3, s_4, s_5, s_6, and s_7 are in $AC(u) \setminus E_2(u)$. For $3 \leq j \leq 7$, $weight_2(s_j, u) = W(s_1, s_2, u_2)/d$. Right: portion of $AC(u_2)$ for the child u_2 of u. $W(s_1, s_2, u_2)$ is equal to the total weight of all segments between s_1 and s_2. If u_2 is a leaf node, then $W(s_1, s_2, u_2)$ equals the total number of segments in $AC(u_2)$ that are situated between s_2 and s_1.

and maintain $AC(u)$. We assume in this section that all segments in every catalog $AC(u)$ are ordered. We can easily order a set $AC_{fl}(u)$ or any set of segments that cross the same vertical line ℓ: the order of segments is determined by (y-coordinates of) intersection points of segments and ℓ. Therefore we will speak of e.g., the largest/smallest segments in such a set.

Element weights. We assign the weight to each element of $AC(u)$ in a bottom-to-top manner: All segments in a set $AC(\ell)$ for every leaf node ℓ are assigned weight 1. Consider a segment $s \in AC_{fl}(u)$, i.e., a segment that spans children u_f, ..., u_l of some internal node u. For $f \leq i \leq l$ let s_1 denote the largest bridge in $E_i(u)$ that is (strictly) smaller than s and let s_2 denote the smallest bridge in $E_i(u)$ that is (strictly) larger than s; we let $W(s_1, s_2, u_i) = \sum_{s_1 < s' < s_2} weight(s', u_i)$, where the sum is over all segments $s' \in AC(u_i)$ and $weight_i(s, u) = W(s_1, s_2, u_i)/d$. See Fig. 3 for an example. We set $weight(s, u) = \sum_{i=f}^{l} weight_i(s, u)$. We keep a weighted search tree for every portion $\mathcal{P}(u)$ of the list $AC(u)$ By a slight misuse of notation this tree will also be denoted by $\mathcal{P}(u)$. Thus every catalog $AC(u)$ is stored in a forest of weighted trees $\mathcal{P}_j(u)$ where every tree corresponds to a portion of $AC(u)$ [4]. We also store a data structure supporting finger searches on $AC(u)$.

Weighted Trees. Each weighted search tree is implemented as a biased (a, b)-tree with parameters $a = B^\delta/2$ and $b = B^\delta$ [10, 19]. The depth of a leaf λ in a biased $(B^\delta/2, B^\delta)$-tree is bounded by $O(\log_B(W/w_\lambda))$, where w_λ is the weight of an element in the leaf λ and W is the total weight of all elements in the tree. Every internal node ν has B^δ children and every leaf holds $\Theta(B)$ segments[5]. In each internal node ν we keep $B^{3\delta}$ segments $\nu.\max_{jk}[i]$. For every child ν_i of ν and for all j and k, $1 \leq j \leq k \leq r$, $\nu.\max_{jk}[i]$ is the highest segment from AC_{jk} in the subtree of ν_i ; if there are no segments from AC_{jk} in the subtree of ν_i, then $\nu.\max_{jk}[i] = \text{NULL}$. Using values of $\nu.\max$ we can find, for any node ν of the biased search tree, the child ν_i of ν that holds the successor segment of the query point q. Hence

[4] In most cases we will omit the subindex and will speak of a weighted tree $\mathcal{P}(u)$ because it will be clear from the context what portion of $AC(u)$ is used.

[5] In the standard biased (a, b)-tree [10, 19], every leaf holds one element. But we can modify it so that every leaf holds $\Theta(B)$ different elements (segments). The weight of a leaf λ is the total weight of all segments stored in λ.

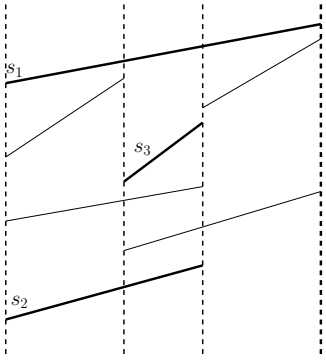

Figure 4 Segments in a group. Segments s_1, s_2, and s_3 are stored in V_2: s_1 and s_2 are the highest and the lowest segments that span the second child u_2; s_3 is a bridge segment from $E_2(u)$.

we can find the smallest segment $n(u)$ in a portion $\mathcal{P}(u)$ that is above a query point q in $O(\log_B(W_P/\omega_n))$ I/Os where W_P is the total weight of all segments in $\mathcal{P}(u)$ and ω_n is the weight of $n(u)$.

Additional Structures. When the segment $n(u)$ is known, we will need to find the bridges that are closest to $n(u)$ in order to continue the search. We keep a list $V_i(u) \subseteq AC(u)$ for each node u and for every i, $1 \leq i \leq r$. $V_i(u)$ contains all segments of $E_i(u)$ and some additional segments chosen as follows: $AC(u)$ is divided into groups so that each group consists of $\Theta(r^6)$ consecutive segments; the only exception is the last group in $AC(u)$ that contains $O(r^6)$ segments (here we use the fact that segments in $AC(u)$ are ordered). We choose the constant in such way that every group but the last one contains $d \cdot r^2$ segments. If a group G contains a segment that spans u_i, then we select the highest segment from G that spans u_i and the lowest segment from G that spans u_i; we store both segments in V_i. See Fig. 4. For every segment in V_i we also store a pointer to its group in $AC(u)$. We keep V_i in a B-tree that supports finger search queries.

Suppose that we know the successor segment $n(u)$ of a query point q in $AC(u)$. We can find the successor segment $b_n(u)$ of q in $E_i(u)$ using V_i: Let G denote the group that contains $n(u)$. We search in G for the segment $b_n(u) \geq n(u)$ using finger search. If $b_n(u)$ is not in G, we consider the highest segment $s_1 \in G$ that spans u_i. By definition of $AC(u)$, there are at most dr^2 segments between $n(u)$ and $b_n(u)$. We can find $b_n(u)$ in $O(\log_B(dr^2)) = O(1)$ I/Os by finger search on V_i using s_1 as the finger. Using a similar procedure, we can find the highest bridge segment $b_p(u) \leq n(u)$ in $E_i(u)$.

Queries. A vertical ray shooting query for a point $q = (q_x, q_y)$ is answered as follows. Let ℓ denote the leaf such that the slab of ℓ contains q. We visit all nodes v_0, v_1, …, v_h on the root-to-leaf path $\pi(\ell)$ where v_0 is the root node and $v_h = \ell$. We find the segment $n(v_i)$ in every visited node, where $n(v_i)$ is the successor segment of q in $AC(v_i)$. Suppose that v_{i+1} is the j-th child of v_i; $n(v_i)$ spans the j-th child of v_i. First we search for $n(v_0)$ in the weighted tree of $AC(v_0)$. Next, using the list V_j, we identify the smallest bridge $b_n(v_0) \in E_j(v_0)$ such that $b_n(v_0) \geq n(v_0)$ and the largest bridge segment $b_p(v_0) \in E_j(v_0)$ such that $b_p(v_0) \leq n(v_0)$. The index j is chosen so that v_1 is the j-th child of v_0. We execute the same operations in nodes v_1, …, v_h. When we are in a node v_i we consider the portion $\mathcal{P}(v_i)$ between bridges $b_p(v_{i-1})$ and $b_n(v_{i-1})$; we search in the weighted tree of $\mathcal{P}(v_i)$ for the successor segment $n(v_i)$ of q. Then we identify the lowest bridge $b_n(v_i) \geq n(v_i)$ and the highest bridge $b_p(v_i) \leq n(v_i)$. When all $n(v_i)$ are computed, we find the lowest segment n^* among $n(v_i)$. Since $\cup_{i=0}^{h} AC(v_i) = \cup_{i=0}^{h} C(v_i)$, n^* is the successor segment of a query point q.

The cost of a ray shooting query can be estimated as follows. Let ω_i denote the weight of $n(v_i)$. Let W_i denote the total weight of all segments of $\mathcal{P}(v_i)$ (we assume that $\mathcal{P}(v_0) = AC(v_0)$). Search for $n(v_i)$ in the weighted tree $\mathcal{P}(v_i)$ takes $O(\log_B(W_i/\omega_i))$ I/Os. By definition of weights, $\omega_i \geq W_{i+1}/d$. Hence

$$\sum_{i=0}^{h} \log_B(W_i/\omega_i) = \log_B W_0 + \sum_{i=0}^{h-1}(\log_B W_{i+1} - \log_B \omega_i) - \log_B \omega_h$$
$$\leq \log_B(W_0/\omega_h) + 2(h+1)\log_B r.$$

We have $\omega_h = 1$ and we will show below that $W_0 \leq n$. Since $r = B^\delta$, $h = O(\log_B n)$ and $\log_B r = O(1)$. Hence the sum above can be bounded by $O(\log_B n)$. When $n(v_i)$ is known, we can find $b_p(v_i)$ and $b_n(v_i)$ in $O(1)$ I/Os, as described above. Hence the total cost of answering a query is $O(\log_B n)$. Since every segment is stored in $O(\log_B n)$ lists $AC(u)$, the total space usage is $O(n \log_B n)$.

It remains to prove that $W_0 \leq n$. We will show by induction that the total weight of all elements on every level of $\mathcal{T}$ is bounded by n: Every element in a leaf node has weight 1; hence their total weight does not exceed n. Suppose that, for some $k \geq 1$, the total weight of all elements on level $k-1$ does not exceed n. Consider an arbitrary node v on level k, let v_1, …, v_r be the children of v, and let m_i denote the total weight of elements in $AC(v_i)$. Every element in $AC(v_i)$ contributes $1/d$ fraction of its weight to at most d different elements in $AC(v)$. Hence $\sum_{e \in AL(v)} weight_i(v) \leq m_i$ and the total weight of all elements in $AC(v)$ does not exceed $\sum_{i=1}^{r} m_i$. Hence, for any level $k \geq 1$, the total weight of $AC(v)$ for all nodes v on level k does not exceed n. Hence the total weight of $AC(u_0)$ for the root node u_0 is also bounded by n.

▶ **Lemma 1.** *There exists an $O(n \log_B n)$-space static data structure that supports point location queries on n non-intersecting segments in $O(\log_B n)$ I/Os.*

The result of Lemma 1 is not new. However we will show below that the data structure described in this section can be dynamized.

4 Semi-Dynamic Ray Shooting for $B \geq \log^8 n$: Main Idea

Now we turn to the dynamic problem. In Sections 4 and 5 we will assume[6] that $B \geq \log^8 n$.

Overview. The main challenge in dynamizing the static data structure from Section 3 is the order of segments. Deletions and insertions of segments can lead to significant changes in the segment order, as explained in Section 2. However segment insertions within a slab are easy to handle in one special case. We will say that a segment $s \in AC(u)$ is a *unit* segment if $s \in AC_{ii}(u)$ for some $1 \leq i \leq r$. In other words a unit segment spans exactly one child u_i of u. Let $L_i(u) = \cup_{f \leq i \leq l} AC_{fl}(u)$ denote the conceptual list of all segments that span u_i. When a unit segment $s \in AC_{ii}(u)$ is inserted, we find the segments s_p and s_n that precede and follow s in $L_i(u)$; we insert s at an arbitrary position in $AC(u)$ so that $s_p < s < s_n$. It is easy to see that the correct order of segments is maintained: the correct order is maintained for the segments that span u_i and other segments are not affected.

[6] Probably a smaller power of log can be used, but we consider $B \geq \log^8 n$ to simplify the analysis.

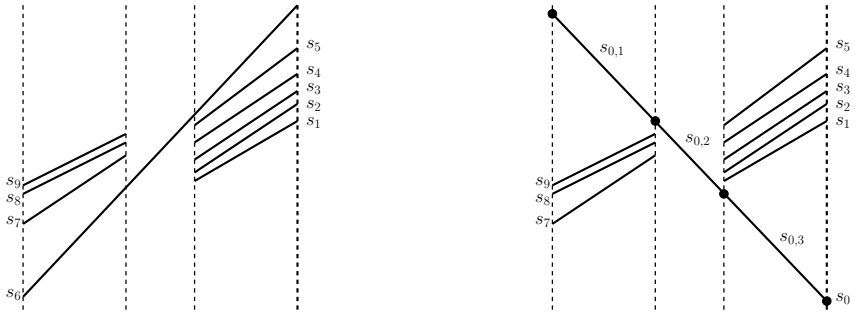

Figure 5 Example from Fig. 2 revisited. Left: original segment order $s_1 \prec s_2 \prec s_3 \prec s_4 \prec s_5 \prec s_6 \prec s_7 \prec s_8 \prec s_9$. Right: segment s_6 is deleted, the new inserted segment s_0 is split into unit segments. The new segment order is e.g., $s_{0,3} \prec s_1 \prec s_2 \prec s_4 \prec s_5 \prec s_{0,2} \prec s_7 \prec s_8 \prec s_9 \prec s_{0,1}$. Thus new unit segments are inserted, but the relative order of other segments does not change.

An arbitrary segment s that is to be inserted into $AC(u)$ can be represented as B^δ unit segments. See Fig. 5 for an example. However we cannot afford to spend B^δ operations for an insertion. To solve this problem, we use bufferization: when a segment is inserted, we split it into B^δ unit segments and insert them into a buffer $\mathcal{B}$. A complete description of the update procedure is given below.

Buffered Insertions. We distinguish between two categories of segments, *old* segments and *new* segments. We know the total order in the set of old segments in the portion $\mathcal{P}(u)$ (and in the list $AC(u)$). New segments are represented as a union of up to r unit segments. When the number of new segments in a portion $\mathcal{P}(u)$ exceeds the threshold that will be specified below, we re-build $\mathcal{P}(u)$: we compute the order of old and new segments and declare all segments in $\mathcal{P}(u)$ to be old.

As explained in Section 3 every portion $\mathcal{P}(u)$ of $AC(u)$ is stored in a biased search tree data structure. Each node of $\mathcal{P}(u)$ has a buffer $\mathcal{B}(\nu)$ that can store up to $B^{3\delta}$ segments. When a new segment is inserted into $\mathcal{P}(u)$, we split it into unit segments and add them to the insertion buffer of ν_r, where ν_r is the root node of $\mathcal{P}(u)$. When the buffer of an internal node ν is full, we *flush* it, i.e., we move all segments from $\mathcal{B}(\nu)$ to buffers in the children of ν. We keep values $\nu.\max_{kj}[i]$, defined in Section 3, for all internal nodes ν. All $\nu.\max_{kl}[\cdot]$ and all segments in $\mathcal{B}(\nu)$ fit into one block of memory; hence we can flush the buffer of an internal node in $O(B^\delta)$ I/Os. When the buffer of an internal node is flushed, we do not change the shape of the tree. When the buffer $\mathcal{B}(\lambda)$ of a leaf node λ is full, we insert segments from $\mathcal{B}(\lambda)$ into the set of segments stored in λ. If necessary we create a new leaf λ' and update the weights of λ and λ'. We can update the biased search tree $\mathcal{P}(u)$ in $O(\log n)$ time. We also update data structures V_i for $i = 1, \ldots, r$. Since a leaf node contains the segments from at most two different groups, we can update all V_i in $O(r)$ I/Os. The biased tree is updated in $O(\log n)$ I/Os. The total amortized cost of a segment insertion into a portion $\mathcal{P}(u)$ is $O(1 + \frac{\log n + r}{B^{3\delta}} + \frac{\log_B n}{B^{2\delta}}) = O(1)$ because $B^\delta > \log n$.

When the number of new segments in $\mathcal{P}(u)$ is equal to $n_{\texttt{old}}/r$, where $n_{\texttt{old}}$ is the number of old segments in $\mathcal{P}(u)$, we rebuild $\mathcal{P}(u)$. Using the method from [8], we order *all* segments in $\mathcal{P}(u)$ and update the biased tree. Sorting of segments takes $O((n_{\texttt{old}}/B)\log_{M/B} n_{\texttt{old}}) = o(n_{\texttt{old}})$ I/Os. We can re-build the weighted tree $\mathcal{P}(u)$ in $O((n_{\texttt{old}}/B^{3\delta})\log n_{\texttt{old}}) = o(n_{\texttt{old}})$ I/Os by computing the weights of leaves and inserting the leaves into the new tree one-by-one.

When a new segment s is inserted, we identify all nodes u_i where s must be stored. For every corresponding list $AC(u_i)$, we find the portion $\mathcal{P}(u_i)$ where s must be stored. This takes $O(\log_B^2 n)$ I/Os in total. Then we insert the trimmed segment s into each portion as described above. The total insertion cost is $O(\log_B^2 n)$. Queries are supported in the same way as in the static data structure described in Section 3. The only difference is that biased tree nodes have associated buffers. Many technical aspects are not addressed in this section. We fill in the missing details and provide the description of the data structure that also supports deletions in Section 5.

5 Ray Shooting for $B \geq \log^8 n$: Fully-Dynamic Structure

Now we give a complete description of the fully-dynamic data structure for vertical ray shooting queries. Deletions are also implemented using bufferization: deleted segments are inserted into deletion buffers $\mathcal{D}(\nu)$ that are kept in the nodes of trees $\mathcal{P}(u)$. Deletion buffers are processed similarly to the insertion buffers. There are, however, a number of details that were not addressed in the previous section. When a new bridge E_i is inserted we need to change weights for a number of segments. When the segment $n(u)$ is found, we need to find the bridges $b_p(u)$ and $b_n(u)$. The complete solution that addresses all these issues is more involved. First, we apply weighted search only to segments from $E(u) = \cup_{i=1}^{r} E_i(u)$. We complete the search and find the successor segment in $AC(u)$ using some auxiliary sets stored in the nodes of $\mathcal{P}(u)$. Second, we use a special data structure to find the bridges $b_p(u)$ and $b_n(u)$. We start by describing the changed structure of weighted trees $\mathcal{P}(u)$.

Segments stored in the leaves of $\mathcal{P}(u)$ are divided into weighted and unweighted segments. Weighted segments are segments from $E(u)$, i.e., weighted segments are used as down-bridges. All other segments are unweighted. Every leaf contains $\Theta(r^2)$ weighted segments. There are at $\Omega(r^2)$ and $O(r^4)$ unweighted segments between any two weighted segments. Hence the total number of segments in a leaf is between $\Omega(r^4)$ and $O(r^6)$. Only weighted segments in a leaf have non-zero weights. Weights of weighted segments are computed in the same way as explained in Section 3. Hence the weight of a leaf λ is the total weight of all weighted segments in λ. The search for a successor of q in $\mathcal{P}(u)$ is organized in such way that it ends in the leaf holding the successor of q in $E(u)$. Then we can find the successor of q in $AC(u)$ using auxiliary data stored in the nodes of $\mathcal{P}(u)$.

We keep the following auxiliary sets and buffers in nodes ν of every weighted tree $\mathcal{P}(u)$. Let $AC_{fl}(u, \nu)$ denote the set of segments from $AC_{fl}(u)$ that are stored in leaf descendants of a node ν.

- **(i)** Sets $\mathrm{Max}_{fl}(\nu)$ and $\mathrm{Min}_{fl}(\nu)$ for all f, l such that $1 \leq f \leq l \leq r$ and for all nodes ν. $\mathrm{Max}_{fl}(\nu)$ ($\mathrm{Min}_{fl}(\nu)$) contains $\min(r^4, |AC_{fl}(u, \nu)|)$ highest (lowest) segments from $AC_{fl}(u, \nu)$. For every segment s in sets $\mathrm{Max}_{fl}(\nu)$ and $\mathrm{Min}_{fl}(\nu)$ we record the index i such that $s \in E_i(u)$ (or NULL if s is not a bridge segment).
- **(ii)** The set $\mathrm{Nav}(\nu)$ for an internal node ν is the union of all sets $\mathrm{Max}_{fl}(\nu_i)$ and $\mathrm{Min}_{fl}(\nu_i)$ for all children ν_i of ν.
- **(iii)** The set $\mathrm{Max}'_{fl}(\nu)$, $1 \leq f \leq l \leq r$ contains highest segments from $AC_{fl}(u, \nu)$ that are not stored in any set $\mathrm{Max}'(u, \mu)$ for an ancestor μ of ν. Either $\mathrm{Max}'_{fl}(\nu)$ holds at least r^4 and at most $2r^4$ segments or $\mathrm{Max}'_{fl}(\nu)$ holds less than r^4 segments and $\mathrm{Max}'_{fl}(\rho)$ for all descendants ρ of ν are empty. In other words, $\mathrm{Max}'_{fl}(\cdot)$ are organized as external priority search trees [6]. The set $\mathrm{Min}'_{fl}(\nu)$ is defined in the same way with respect to the lowest segments. We use Max' and Min' to maintain sets Max and Min.
- **(iv)** Finally we keep an insertion buffer $\mathcal{B}(\nu)$ and a deletion buffer $\mathcal{D}(\nu)$ in every node ν.

Deletions. If an old segment s is deleted, we insert it into the deletion buffer $\mathcal{D}(\nu_R)$ of the root node ν_R. If a new segment s is deleted, we split s into $O(r)$ unit segments and insert them into $\mathcal{D}(\nu_R)$. When one or more segments are inserted into $\mathcal{D}(\nu_r)$, we also update sets $\mathrm{Max}_{fl}(\nu_R)$ and $\mathrm{Min}_{fl}(\nu_R)$. For any node $\nu \in \mathcal{P}(u)$, when the number of segments in $\mathcal{D}(\nu)$ exceeds r^3, we flush both $\mathcal{D}(\nu)$ and $\mathcal{B}(\nu)$ using the following procedure. First we identify segments $s \in \mathcal{B}(\nu) \cap \mathcal{D}(\nu)$ and remove such s from both $\mathcal{B}(\nu)$ and $\mathcal{D}(\nu)$. Next we move segments from $\mathcal{B}(\nu)$ and $\mathcal{D}(\nu)$ to buffers $\mathcal{B}(\nu_i)$ and $\mathcal{D}(\nu_i)$ in the children ν_i of ν. For every child ν_i of ν, first we update sets $Max'_{fl}(\nu_i)$ by removing segments from $\mathcal{D}(\nu_i)$ (resp. inserting segments from $\mathcal{B}(\nu_i)$) if necessary. Then we take care that the size of $\mathrm{Max}'_{fl}(\nu_i)$ is not too small. If some $\mathrm{Max}'_{fl}(\nu_i)$ contains less than r^4 segments and more than 0 segments, we move up segments from the children of ν_i into ν_i, so that the total size of $\mathrm{Max}'_{fl}(\nu_i)$ becomes equal to $2r^4$ or all segments are moved from the corresponding sets $\mathrm{Max}'_{fl}(\cdot)$ in the children of ν_i into $\mathrm{Max}'_{fl}(\nu_i)$. We recursively update $\mathrm{Max}'_{fl}(\cdot)$ in each child of ν_i using the same procedure.

Next, we update sets $\mathrm{Max}_{fl}(\nu_i)$. We compute $\mathcal{M}_{fl} = \cup \mathrm{Max}'_{fl}(\mu)$ where the union is taken over all proper ancestors μ of ν. Every segment in $\mathrm{Max}_{fl}(\nu)$ is either from $\mathrm{Max}'_{fl}(\nu)$ or from $\mathrm{Max}'_{fl}(\mu)$ for a proper ancestor μ of ν. Hence we can compute all $\mathrm{Max}_{fl}(\nu_i)$ when $\mathcal{M}_{fl}$ and $\mathrm{Max}'_{fl}(\nu_i)$ are known. Sets $\mathrm{Min}'_{fl}(\nu_i)$ and $\mathrm{Min}_{fl}(\nu_i)$ are updated in the same way. Finally we update the set $\mathrm{Nav}(\nu)$ by collecting segments from $\mathrm{Max}_{fl}(\nu_i)$ and $\mathrm{Min}_{fl}(\nu_i)$.

All segments needed to re-compute sets after flushing buffers $\mathcal{D}(\nu)$ and $\mathcal{B}(\nu)$ fit into one block of space. Hence we can compute the set $\mathcal{M}$ in $O(\log_B n) = O(r)$ I/Os and all sets in each node ν_i in $O(1)$ I/Os. The set $\mathrm{Nav}(\nu)$ is updated in $O(r)$ I/Os. Since each node has $O(r)$ children, the total number of I/Os needed to flush a buffer is $O(r)$. Every segment can be divided into up to r unit segments and each unit segment can contribute to $\log_B n$ buffer flushes. Hence the total amortized cost per segment is $O(\frac{r^2 \log_B n}{r^3}) = O(1)$. We did not yet take into account the cost of refilling the buffers Max'; using the analysis similar to the analysis in [12, Section 4], we can estimate the cost of re-filling Max' as $O(\frac{\log_B n}{r^3}) = o(1)$.

We do not store buffers in the leaf nodes. Let $S(\lambda)$ be the set of segments kept in a leaf λ and let $S_W(\lambda)$ be the set of weighted segments stored in λ. When we move segments from $\mathcal{B}(\nu)$ or $\mathcal{D}(\nu)$ to its leaf child λ, we update $S(\lambda)$ accordingly. This operation changes the weight of λ. Hence we need to update the weighted tree $\mathcal{P}(u)$ in $O(\log n)$ I/Os. Sets $\mathrm{Max}_{fl}(\cdot)$ and $\mathrm{Min}_{fl}(\cdot)$ are also updated.

After an insertion of new segments into a leaf node, we may have to insert or remove some bridges in $E_i(u)$ for $1 \leq i \leq r$. When we insert a new bridge b into $E_i(u)$, we must split some portion $\mathcal{P}(u_i)$ into two new portions, $\mathcal{P}_1(u_i)$ and $\mathcal{P}_2(u_i)$. Additionally we must change the weights of the bridge segments in $E_i(u)$ that precede and follow b. The cost of splitting $\mathcal{P}(u_i)$ is $O(\log n)$. We also need $O(\log n)$ I/Os to change the weights of two neighbor bridges. Hence the total cost of inserting a new bridge is $O(\log n)$. We insert a bridge at most once per $O(r)$ insertions into $AC(u)$ because every new segment is divided into up to r unit segments. We remove a bridge at most once after $O(r)$ deletions. See [13] for the description of the method to maintain bridges in catalogs $AC(u)$. Thus the total amortized cost incurred by a bridge insertion or deletion is $O(\frac{\log n}{r}) = O(1)$.

Insertions. Insertions are executed in a similar way. A new inserted segment is split into $O(r)$ unit segments that are inserted into the buffer $\mathcal{B}(\nu_R)$ for the root node ν_R. The buffers and auxiliary sets are updated and flushed in the same way as in the case of deletions. When the number of new segments in some portion $\mathcal{P}(u)$ is equal to $n_{\mathtt{old}}/r$, where $n_{\mathtt{old}}$ is the number of old segments in $\mathcal{P}(u)$, we rebuild $\mathcal{P}(u)$. As explained in Section 4, rebuilding of $\mathcal{P}(u)$ incurs an amortized cost of $o(1)$.

Queries. The search for the successor segment $n(u)$ in the weighted tree $\mathcal{P}(u)$ consists of two stages. Suppose that the query point q is in the slab of the i-th child u_i of u. First we find the successor $b_n(u)$ of q in $E_i(u)$ by searching in $\mathcal{P}(u)$. We traverse the path from the root to the leaf λ_n holding $b_n(u)$. In every node ν we select its leftmost child ν_j, such that $\mathrm{Max}_{fl}(\nu_j)$ for some $f \le i \le l$ contains a segment s that is above q and s is not deleted (i.e., $s \notin \mathcal{D}(\mu)$ for all ancestors μ of ν). The size of each set $\mathrm{Max}_{fl}(\nu_k)$ is larger than the total size of all $\mathcal{D}(\mu)$ in all ancestors μ of ν. Hence every $\mathrm{Max}_{fl}(\nu_i)$ contains some elements that are not deleted unless the set $C_{fl}(u, \nu_i)$ is empty. Therefore we select the correct child ν_j in every node. Since $\mathcal{P}(u)$ is a biased search tree [10, 19], the total cost of finding the leaf λ_n is bounded by $O(\log(W_P/\omega_\lambda)) = O(\log(W_P/\omega_n))$ where ω_λ is the total weight of all segments in λ_n and $\omega_n \le \omega_\lambda$ is the weight of the bridge segment $b_n(u)$.

During the second stage we need to find the successor segment $n(u)$ of q in $AC(u)$. The distance between $n(u)$ and $b_n(u)$ in $AC(u)$ can be arbitrarily large. Nevertheless $n(u)$ is stored in one of the sets $\mathrm{Nav}(\mu)$ for some ancestor μ of λ_n. Suppose that $n(u)$ is an unweighted segment stored in a leaf λ' of $\mathcal{P}(u)$ and let μ denote the lowest common ancestor of λ and λ'. Let μ_k be the child of μ that is an ancestor of λ'. There are at most r^4 segments in $AC_{fl}(u)$ between $n(u)$ and $b_n(u)$. Hence, $n(u)$ is stored in the set $\mathrm{Max}_{fl}(\mu_k)$. Hence, $n(u)$ is also stored in $\mathrm{Nav}(\mu)$. We visit all ancestors μ of λ_n and compute $\mathcal{D} = \cup_\mu \mathcal{D}(\mu)$. Then we visit all ancestors one more time and find the successor of q in $\mathrm{Nav}(\mu) \setminus \mathcal{D}$. The asymptotic query cost remains the same because we only visit the nodes between λ_n and the root and each node is visited a constant number of times.

We need to consider one additional special case. It is possible that there are no bridge segments $s \in E_i(u)$ stored in the leaves of $\mathcal{P}(u)$. In this case there are at most r^2 segments in $AC_{fl}(u)$ for every pair f, l, satisfying $f \le i \le l$, stored in the leaves of $\mathcal{P}(u)$. For each portion $\mathcal{P}(u)$, if there are at most r^2 segments in $AC_{fl}(u) \cap \mathcal{P}(u)$, we keep the list of all such segments. All such lists fit into one block of memory. We also keep the list of indexes i, such that $E_i(u) \cap \mathcal{P}(u)$ is empty.Suppose that we need to find the successor of q and $\mathcal{P}(u) \cap E_i(u)$ is empty. Then we simply examine all segments in $AC_{fl}(u) \cap \mathcal{P}(u)$ for all $f \le i \le l$ and find the successor of q in $O(1)$ I/Os.

When $n(u)$ is known, we need to find $b_p(u)$ and $b_n(u)$, if $b_n(u)$ was not computed at the previous step. It is not always possible to find these bridges using $\mathcal{P}(u)$ because $b_p(u)$ and $b_n(u)$ can be outside of $\mathcal{P}(u)$. To this end, we use the data structure for colored union-split-find problem on a list (list-CUSF) that will be described in the full version of this paper [24]. We keep the list $V(u)$ containing all down-bridges from $E_i(u)$, for $1 \le i \le r$, and all up-bridges from $UP(u)$. Each segment in $e \in V(u)$ is associated to an interval; a segment $e \in V_i(u)$ is associated to an interval $[i, i]$ and a segment from $UP(u)$ is associated to a dummy interval $[-1, -1]$. For any segment $e \in V(u)$ we can find the preceding/following segment associated to an interval $[i, i]$ for any i, $1 \le i \le r$, in $O(\log\log_B n)$ I/Os. Updates of $V(u)$ are supported in $O(\log\log_B n)$ I/Os. Since we insert or remove bridge segments once per r^2 updates, the amortized cost of maintaining the list-CUSF structure is $O(1)$.

Summing up. By the same argument as in Section 3, weighted searches in all nodes take $O(\log_B n)$ I/Os in total. Additionally we spend $(\log\log_B n)$ I/Os in every node with a query to list-CUSF. Thus the total query cost is $O(\log_B n \log\log_B n)$. When a segment is deleted, we remove it from $O(\log_B n)$ lists $AC(u)$ and from secondary structures (weighted trees etc.) in these nodes. The deletions take $O(1)$ I/Os per node or $O(\log_B n)$ I/Os in total. When a segment is inserted, it must be inserted into $O(\log_B n)$ lists $AC(u)$. We first have to spend $O(\log_B n)$ I/Os to find the portion $\mathcal{P}(u)$ of each $AC(u)$ where it must be stored. When $\mathcal{P}(u)$

is known, an insertion takes $O(1)$ amortized I/Os as described above. The total cost of an insertion is $O(\log_B^2 n)$ I/Os. Since every segment is stored in $O(\log_B n)$ lists, the total space is $O(n \log_B n)$.

▶ **Lemma 2.** *If $B > \log^8 n$, then there exists an $O(n \log_B n)$ space data structure that supports vertical ray shooting queries on a dynamic set of n non-intersecting segments in $O(\log_B n \log\log_B n)$ I/Os. Insertions and deletions of segments are supported in $O(\log_B^2 n)$ and $O(\log_B n)$ amortized I/Os respectively.*

6 Faster Insertions

When a new segment s is inserted into our data structure, we need to find the position of s in $O(\log_B n)$ lists $AC(u)$ (to be precise, we need to know the portion $\mathcal{P}(u)$ of $AC(u)$ that contains s). When positions of s in $AC(u)$ are known, we can finish the insertion in $O(\log_B n)$ I/Os. In order to speed-up insertions, we use the multi-colored segment tree of Chan and Nekrich [13]. Segments in lists $C(u)$ are assigned colors χ, so that the total number of different colors is $O(\log H)$ where $H = O(\log_B n)$ is the height of the segment tree. Let $C_\chi(u)$ denote the set of segments of color χ in $C(u)$. We apply the technique of Sections 3- 5 to each color separately. That is, we create augmented lists $AC_\chi(u)$ and construct weighted search trees $\mathcal{P}_\chi(u)$ for each color separately. The query cost is increased by factor $O(\log H)$, the number of colors. The deletion cost is also increased by $O(\log H)$ factor because we update the data structure for each color separately. When a new segment s is inserted, we insert it into some lists $AC_{\chi_i}(u_i)$ where u_i is the node such that s spans u_i but does not span its parent and χ_i is some color (the same segment can be assigned different colors χ_i in different nodes u_i). We can find the position of s in all $AC_{\chi_i}(u_i)$ with $O(\log_B n \log H + H \cdot t_{\texttt{usf}}) = O(\log_B n \log\log_B n)$ I/Os where $t_{\texttt{usf}} = O(\log\log_B n)$ is the query cost in a union-split-find data structure in the external memory model. See [13] for a detailed description.

▶ **Lemma 3.** *If $B > \log^8 n$, then there exists an $O(n \log_B n)$ space data structure that supports vertical ray shooting queries on a dynamic set of non-intersecting segments in $O(\log_B n (\log\log_B n)^2)$ I/Os. Insertions and deletions of segments can be supported in $O(\log_B n \log\log_B n)$ amortized I/Os.*

7 Missing Details

Using the method from [13] we can reduce the space usage of our data structure to linear at the cost of increasing the query and update complexity by $O(\log\log_B n)$ factor.The resulting data structure supports queries in $O(\log_B n (\log\log_B n)^2)$ I/Os and updates in $O(\log_B n (\log\log_B n)^3)$ amortized I/Os. Details will be provided in the full version [24].

In our exposition we assumed for simplicity that the tree $\mathcal{T}$ does not change, i.e., the set of x-coordinates of segment endpoints is fixed and known in advance. To support insertions of new x-coordinate, we can replace the static tree $\mathcal{T}$ with a weight-balanced tree with node degree $\Theta(r) = \Theta(B^\delta)$. We also assumed that the block size B is large, $B > \log^8 n$. If $B \leq \log^8 n$, the linear-space internal memory data structure [13] achieves $O(\log n (\log\log n)^2) = O(\log_B n (\log\log_B n)^3)$ query cost and $O(\log n \log\log n) = O(\log_B n (\log\log_B n)^2)$ update cost because $\log n = O(\log_B n \log\log_B n)$ and $\log\log n = O(\log\log_B n)$ for $B \leq \log^8 n$. Thus we obtain our main result.

▶ **Theorem 4.** *There exists an $O(n)$ space data structure that supports vertical ray shooting queries on a dynamic set of n non-intersecting segments in $O(\log_B n(\log\log_B n)^3)$ I/Os. Insertions and deletions of segments are supported in $O(\log_B n(\log\log_B n)^2)$ amortized I/Os.*

References

1 Pankaj K. Agarwal, Lars Arge, Gerth Stølting Brodal, and Jeffrey Scott Vitter. I/O-efficient dynamic point location in monotone planar subdivisions. In *Proc. 10th Annual ACM-SIAM Symposium on Discrete Algorithms, (SODA)*, pages 11–20, 1999.

2 Alok Aggarwal and Jeffrey Scott Vitter. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM*, 31(9):1116–1127, 1988. doi:10.1145/48529.48535.

3 Lars Arge, Gerth Stølting Brodal, and Loukas Georgiadis. Improved dynamic planar point location. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, pages 305–314, 2006. doi:10.1109/FOCS.2006.40.

4 Lars Arge, Gerth Stølting Brodal, and S. Srinivasa Rao. External Memory Planar Point Location with Logarithmic Updates. *Algorithmica*, 63(1-2):457–475, 2012. doi:10.1007/s00453-011-9541-2.

5 Lars Arge, Andrew Danner, and Sha-Mayn Teh. I/O-efficient point location using persistent B-trees. *ACM Journal of Experimental Algorithmics*, 8, 2003. doi:10.1145/996546.996549.

6 Lars Arge, Vasilis Samoladas, and Jeffrey Scott Vitter. On Two-Dimensional Indexability and Optimal Range Search Indexing. In *Proc. 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 346–357, 1999. doi:10.1145/303976.304010.

7 Lars Arge and Jan Vahrenhold. I/O-efficient dynamic planar point location. *Computational Geometry*, 29(2):147–162, 2004. doi:10.1016/j.comgeo.2003.04.001.

8 Lars Arge, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-Memory Algorithms for Processing Line Segments in Geographic Information Systems. *Algorithmica*, 47(1):1–25, 2007. doi:10.1007/s00453-006-1208-z.

9 Hanna Baumgarten, Hermann Jung, and Kurt Mehlhorn. Dynamic point location in general subdivisions. *J. Algorithms*, 17(3):342–380, 1994. doi:10.1006/jagm.1994.1040.

10 Samuel W. Bent, Daniel Dominic Sleator, and Robert Endre Tarjan. Biased Search Trees. *SIAM J. Comput.*, 14(3):545–568, 1985. doi:10.1137/0214041.

11 Jon Louis Bentley. Algorithms for Klee's rectangle problems. Unpublished manuscript, Department of Computer Science, Carnegie-Mellon University, 1977.

12 Gerth Stølting Brodal. External Memory Three-Sided Range Reporting and Top-k Queries with Sublogarithmic Updates. In *Proc. 33rd Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 23:1–23:14, 2016. doi:10.4230/LIPIcs.STACS.2016.23.

13 Timothy M. Chan and Yakov Nekrich. Towards an Optimal Method for Dynamic Planar Point Location. In *Proc. 56th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 390–409, 2015. doi:10.1109/FOCS.2015.31.

14 Timothy M. Chan and Konstantinos Tsakalidis. Dynamic Planar Orthogonal Point Location in Sublogarithmic Time. In Bettina Speckmann and Csaba D. Tóth, editors, *34th International Symposium on Computational Geometry (SoCG 2018)*, volume 99 of *LIPIcs*, pages 25:1–25:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018. doi:10.4230/LIPIcs.SoCG.2018.25.

15 Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(2):133–162, 1986. doi:10.1007/BF01840440.

16 Siu-Wing Cheng and Ravi Janardan. New results on dynamic planar point location. *SIAM J. Comput.*, 21(5):972–999, 1992. doi:10.1137/0221057.

17 Yi-Jen Chiang, Franco P. Preparata, and Roberto Tamassia. A unified approach to dynamic point location, ray shooting, and shortest paths in planar maps. *SIAM J. Comput.*, 25(1):207–233, 1996. doi:10.1137/S0097539792224516.

18 Yi-Jen Chiang and Roberto Tamassia. Dynamization of the trapezoid method for planar point location in monotone subdivisions. *Int. J. Comput. Geometry Appl.*, 2(3):311–333, 1992. doi:10.1142/S0218195992000184.

19 Joan Feigenbaum and Robert Endre Tarjan. Two New Kinds of Biased Search Trees. *Bell Systems Technical Journal*, 62(10):3139–3158, 1983.

20 Yoav Giora and Haim Kaplan. Optimal dynamic vertical ray shooting in rectilinear planar subdivisions. *ACM Transactions on Algorithms*, 5(3):28, 2009. doi:10.1145/1541885.1541889.

21 Michael T. Goodrich and Roberto Tamassia. Dynamic trees and dynamic point location. *SIAM J. Comput.*, 28(2):612–636, 1998. doi:10.1137/S0097539793254376.

22 Michael T. Goodrich, Jyh-Jong Tsay, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-Memory Computational Geometry (Preliminary Version). In *Proc. 34th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 714–723, 1993. doi:10.1109/SFCS.1993.366816.

23 Kurt Mehlhorn and Stefan Näher. Dynamic fractional cascading. *Algorithmica*, 5(2):215–241, 1990. doi:10.1007/BF01840386.

24 J. Ian Munro and Yakov Nekrich. Dynamic Planar Point Location in External Memory. *CoRR*, abs/1903.06601, 2019. arXiv:1903.06601.

25 Franco P. Preparata and Roberto Tamassia. Fully dynamic point location in a monotone subdivision. *SIAM J. Comput.*, 18(4):811–830, 1989. doi:10.1137/0218056.

26 Franco P. Preparata and Roberto Tamassia. Efficient point location in a convex spatial cell-complex. *SIAM J. Comput.*, 21(2):267–280, 1992. doi:10.1137/0221020.